Parallel Lint

Andrey Karpov
OOO "Program Verification Systems"

June 2009

Abstract
Introduction
PC-Lint tool
VivaMP tool
Static analysis implemented in Intel C++ compiler
Conclusion
References

Abstract

The article describes a new direction in development of static code analyzers - verification of parallel programs. The article reviews several static analyzers which can claim to be called "Parallel Lint".

Introduction

The article is intended for developers of Windows-applications using C/C++ languages but may be also useful for many people interested in the issues of static code analysis.

Static code analysis is a process of software verification without actual execution of programs being examined. Usually it is the source code of the program that is directly analyzed but sometimes one type of object code is analyzed [1].

Static analysis is a kind of software testing with the help of code review. But while in case of code review the code being tested must be looked through by a person, in case of static analysis specialized software called a static analyzer takes part of the work. A static analyzer detects potentially unsafe places in the program code and suggests the user perform further analysis and correction of these places. It helps reduce the amount of work of code review.

One of the first static analyzers and the most popular one was Lint utility which appeared in 1979 as part of Unix 7 distribution kit as the main tool of controlling the quality of software written in C [2, 3]. This tool is so popular that the word "lint" has nearly become a synonym of "static analyzer". And very often you can see not "a static analysis tool" but "a lint-like tool".

Some time later new powerful means of static analysis, both general-purpose and specialized, appeared: Coverity Prevent, PC-Lint, KlocWork K7, PolySpace, Viva64, FXCop, C++Test, JLint, UNO etc.

These static analyzers detect a lot of errors varieties of which are even impossible to list. They are constantly modified and they are learning to detect more and more types of errors. One of such new directions is verification of parallel programs.

As multi-core microprocessors appeared parallel programming exceeded the limits of highly tailored solutions and spread quickly in the sphere of various applications. Parallel programming requires specialized tools for creating, testing and debugging programs. And while in the sphere of technology of creating parallel applications everything is rather good, in the sphere of verification and testing of these

applications there are a lot gaps. It is this that urges developers of static analyzers to develop this territory which is rather new for them - the territory of verification of parallel applications. Now it's the time of new solutions. Now it's the time of "Parallel Lint".

Parallel Lint is not a name of a concrete tool. But it is a good phrase which describes simply and briefly the new class of tools for analyzing parallel code. In this article the reader will get acquainted with some tools in which diagnosing of the new class of parallel errors is implemented and which can be justly called "Parallel Lint".

PC-Lint tool

In the 9th version of Gimpel Software PC-lint diagnosing of errors in parallel programs built on POSIX threads and similar technologies is implemented. In other words PC-Lint tool is not focused on some particular parallel technologies and can be adapted by a user for the programs built on various libraries implementing parallelism. This flexibility is achived due to arrangement of auxiliary comments for PC-Lint in the code and its settings. As usual, the price of this flexibility is a higher complexity.

Still, complex setting is needed only if you use a parallel technology unknown to PC-Lint. PC-Lint supports POSIX threads and can detect errors relating to locks by itself if such functions as pthread_mutex_lock() and pthread_mutex_unlock() are used.

But anyway PC-Lint requires some hints from the programmer's side and is helpless without them. The point is that the static analyzer does not know if some code section is executed in parallel mode and can only make some suggestions. It is not so easy to understand whether this is a parallel call or not. The call can depend in a complex way on the logic of the algorithm and input data which the static analyzer lacks.

Let's consider an example of the function (given in the manual to PC-Lint 9.0):

```
void f()
{
  static int n = 0;
  /* ... */
}
```

The problem is that if f() function is called from parallel threads an error of initializing 'n' variable may occur. To diagnose this case we need to explicitly define to PC-Lint analyzer that f() function can be called in parallel mode. To do so we need to use a consturcion like:

```
//lint -sem(f, thread)
```

Then, when diagnosing the code, you will see the warning: Warning 457: "Thread 'f(void)' has an unprotected write access to variable 'n' which is used by thread 'f(void) .

If the paralleliling mechanism is built not on POSIX threads you will have to use special additional directives to hint PC-Lint what functions cause locks and unlocks:

```
-sem(function-name, thread_lock)
-sem(function-name, thread_unlock)
```

In this case you can detect errors in the code like the following one:

```
//lint -sem( lock, thread_lock )
//lint -sem( unlock, thread_unlock )
```

```
extern int g();
void lock(void), unlock(void);
void f()
{
  //-------------
  lock();
  if( g() )
    return; // Warning 454
  unlock();
  //-------------
  if( g() )
  {
    lock();
    unlock();
    unlock(); // Warning 455
    return;
  }
  //-------------
  if( g() )
    lock();
  { // Warning 456
    // do something interesting
  }
}
```

PC-Lint analyzer can efficiently find errors relating to parallelism if you give the necessary 'hints' to it beforehand. Otherwise, it can perform analysis of a parallel code as of a sequential one and therefore fail to detect some errors.

Despite the necessity of additional work of arranging the hints, PC-Lint tool is a very convenient and powerful tool able to detect many parallel errors. It can be especcialy useful when used together with Visual Lint shell providing a more convenient user interface to this analyzer.

To learn more about diagnostic abilities of PC-Lint 9.0 see the manual to PC-lint/FlexeLint 9.0 Manual Excerpts [4].

Among disadvantages of this tool we can mention its inability to diagnose parallel code built on OpenMP technology. It does not process OpenMP directives and hints will not help here. But there are other static analyzers for diagnosing OpenMP programs and we will speak about them further [5].

VivaMP tool

VivaMP is a powerful specialized tool intended for verification of the code of applications built on OpenMP technology. This tool has been initially developed with the purpose of testing OpenMP parallel code and that's why it can claim to be called "Parallel Lint" more than all the others.

The analyzer integrates into Visual Studio 2005/2008 environment and allows you to start work immediately without complicated settings or arrangement of comments in the program code. It is a great advantage of the tool for it allows you to try and master the product easily. Like any other static analyzer VivaMP will reuqire additional setting while operating it to reduce the number of false responses. But this is an inevitable sacrifice all the static analyzers demand.

The possibility to avoid tiring setting of the analyzer is explained by the peculiarities of OpenMP technology. Description of OpenMP directives is in itself an explaining comment for the analyzer providing information about the program's structure: which part is executed in parallel mode, which resources are local and which are general etc. All this allows the analyzer to perform rather a thorough analysis without demanding assistance of the programmer.

Let's show the principles of VivaMP operation on several simple examples.

Example N1.

```
#pragma omp parallel for
for (size_t i = 0; i != n; ++i)
{
  float *array =
    new float[10000]; // V1302
  delete [] array;
}
```

In this code, VivaMP analyzer diagnoses the following error: "V1302. The 'new' operator cannot be used outside of a try..catch block in a parallel section.". The error relates to throwing of an exception from a parallel block. According to OpenMP specification, if you use exceptions inside a parallel block all these exceptions must be processed inside this block. If you use 'new' operator inside parallel code, you must provide catching of the exception which will be generated according to C++ standard when an error of memory allocation occurs.

This example leads to incorrect program behavior and most likely to program crash if an error of memory allocation occurs.

Correction of the code lies in processing of exceptions inside the parallel block and transferring of information about the error through other mechanisms or refusing to use 'new' operator.

The following corrected code is safe from the viewpoint of VivaMP analyzer:

```
#pragma omp parallel for
for (size_t i = 0; i != n; ++i)
{
  try {
    float *array =
      new float[10000]; // OK
    delete [] array;
  }
  catch (std::bad_alloc &) {
    // process exception
  }
}
```

Example N2.

```
int a = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100000; i++)
{
    a++; //  V1205
}
```

This is an example of a typical error of race condition. This is a programming error of a multi-task system where operation of the system depends on the order in which the code parts are executed. Race condition appears when several threads of a multi-thread application try to get access to data simultaneously and one thread is performing writing. Race conditions can lead to unexptected results and are very often difficult to detect. Sometimes the consequences of the race condition can occur only in a very large time period and in quite a different place of the application. To avoid race conditions synchronization means are used which allow you to arrange operations executed by different threads in a right way.

In this code VivaMP analyzer diagnoses the following error: "V1205. Data race risk. Unprotected concurrent operation with the "a" variable". As all the threads are writing into the same memory space and reading from it simultaneously, the value of the variable after this cycle cannot be predicted. To make this operation safe you must put it into a critical section or (for in this example the operation is elementary) use "#pragma omp atomic" directive:

```
int a = 0;
#pragma omp parallel for num_threads(4)
for (int i = 0; i < 100000; i++)
{
    #pragma omp atomic
    a++; // OK
}
```

To learn more about type errors occurring when developing OpenMP applications see the article "32 OpenMP traps for C++ developers" [6]. Diagnosing of most of the errors described in the article is already implemented in VivaMP analyzer or will appear in new versions.

Static analysis implemented in Intel C++ compiler

Diagnosing of parallel OpenMP errors is implemented in Intel C++ 11.0 compiler. Guess how this new mechanism is called in the compiler. Right, the new function of the compiler is called "Parallel Lint" [7].

The static analyzer embedded into Intel C++ compiler allows you to diagnose various synchronization errors, race conditions etc. For this you must define the key /Qdiag-enable:sc-parallel{1|2|3} to the compiler, where numbers define the level of analysis. An additional parameter which is important when analyzing parallel programs is the compiler's key /Qdiag-enable:sc-include which points to the compiler to search errors in header files as well.

In the mode of testing a parallel program the compiler does not generate the code being executed. At the stage of compilation special pseudo-object files (*.obj) are created which contain data structures including information necessary for analysis instead of object code. Then these files are sent to the input of the static analyzer which performs analysis of the parallel code.

As it was said above, the executed code is not generated when setting the compiler for static analysis of parallel OpenMP programs. That's why it would be a convenient solution to keep separate configurations for building of a project and for its static analysis.

As a result, the developer can, for example, detect an error in the following code:

```
#include <stdio.h>
#include "omp.h"

int main(void)
{
  int i;
  int factorial[10];

  factorial[0]=1;
  #pragma omp parallel for
  for (i=1; i < 10; i++) {
   factorial[i] = i * factorial[i-1]; // warning #12246
  }

  return 0;
}
```

In static analysis mode Intel C++ compiler will show the following warning: "omp.c(13): warning #12246: flow data dependence from (file:omp.c line:13) to (file:omp.c line:13), due to "factorial" may lead to incorrect program execution in parallel mode".

To learn more about the technology of Intel C++ static analysis visit Dmitriy Petunin's webinar "Static Analysis and Intel® C/C++ Compiler ("Parallel Lint" overview)" which can be found in the archives on Intel site [8].

As you see, functionality of Intel C++ static analyzer is similar to VivaMP's. I cannot say which one is better and can be useful for your tasks. On the one hand the analyzer in Intel C++ is a tool you get together with the compiler. On the other hand there are a lot of various functions in Intel C++ compiler and "Parallel Lint" is only one of them while VivaMP analyzer is a highly tailored product which is rapidly developing.

Most likely, VivaMP analyzer would be useful for those developers who use Visual C++ as it allows performing analysis without any changes in a project's configurations and source code. To use Intel C++ abilities the developers will have to adapt their project for building it by this complier at first. That's why "Parallel Lint" in Intel C++ would be useful first of all for those developers who have been already using Intel C++ compiler to build their applications.

Conclusion

Of course, there are other tools which can claim to be called "Parallel Lint". But these tools are mostly dynamic analyzers or combine static and dynamic analysis. So it would be incorrect to describe them in this article. The readers can get acquainted with some of them in the article "Tools And Techniques to Identify Concurrency Issues" [9].

And we wish all our readers safe parallel code!

References

1. Wikipedia. Static code analysis. http://www.viva64.com/go.php?url=12
2. Ian F. Darwin. Checking C Programs with lint, O'Reilly, 1988, ISBN 0-937175-30-7.
3. Wikipedia. Lint. http://www.viva64.com/go.php?url=225
4. PC-lint/FlexeLint 9.0 Manual Excerpts. http://www.viva64.com/go.php?url=226
5. Peaceful coexistence of PC-Lint and VivaMP. http://www.viva64.com/blog/en/2009/02/23/31/
6. Alexey Kolosov, Andrey Karpov, Evgeniy Ryzhkov. 32 OpenMP traps for C++ developers. http://www.viva64.com/art-3-2-1023467288.html
7. David Worthington. Intel adds parallel programming features to compilers. http://www.viva64.com/go.php?url=227
8. Intel Software Network. Forums. Thread "Parallel Lint". http://www.viva64.com/go.php?url=228
9. Rahul V. Patil and Boby George. Tools And Techniques to Identify Concurrency Issues. http://www.viva64.com/go.php?url=132
